MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs

Hongxu Chen, Shengjian Guo, Yinxing Xue, Yuelei Sui
Cen Zhang, Yuekang Li, Haijun Wang, Yang Liu

Background

Bugs/vulnerabilities in multithreaded programs are subtle to be detected

Many programs rely on specific test inputs to trigger multithreading-relevant bugs

Existing fuzzing techniques cannot effectively generate multithreading-relevant tests

Motivation (1) – The problem

```c
int g_var = -1;
void modify(int *pv) { *pv -= 2;}                   // 9

void check(char * buf) {
  if (is_invalid(buf)) { exit(1); }
  else { modify((int*)buf); }
}

char* compute(void *s_var) {
  g_var += 1;                                       // 1
  g_var *= 2;                                       // 2
  if ((int*)s_var[0]<0)                             // 3
      modify((int*)s_var);                          // 4
  pthread_mutex_lock(&m);                           // 5
  modify(&g_var);                                   // 6
  pthread_mutex_unlock(&m);                         // 7
  return (char*)s_var;                              // 8
}

int main(int argc, char **argv) {
  char * buf = read_file_content(argv[1]);
  check(buf);
  pthread_t T1, T2;
  pthread_create(T1,NULL,compute,buf);
  pthread_create(T2,NULL,compute,buf+128);
  ......
}
```

- Coverage depends on test inputs

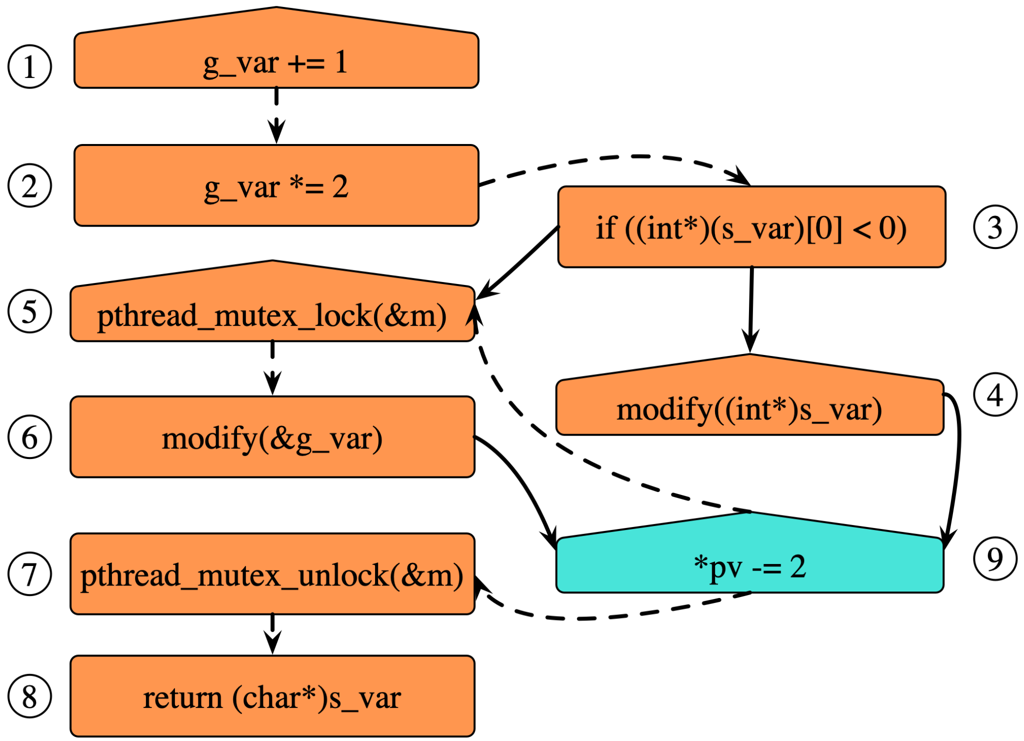
e.g., the program may or may not execute 4 according to the condition of 3, purely dependent on inputs

- Coverage depends on thread-scheduling

e.g., 1 :“g_var+=1” 2 : “g_var*=2”

- T1: 1 →T2: 1 →T2: 2 →T1: 2 ➔ g_var=4
- T1: 1 →T2: 1 →T1: 2 →T2: 2 ➔ g_var=4
- T1: 1 →T1: 2 →T2: 1 →T2: 2 ➔ g_var=2

Motivation (2) – Existing Solutions

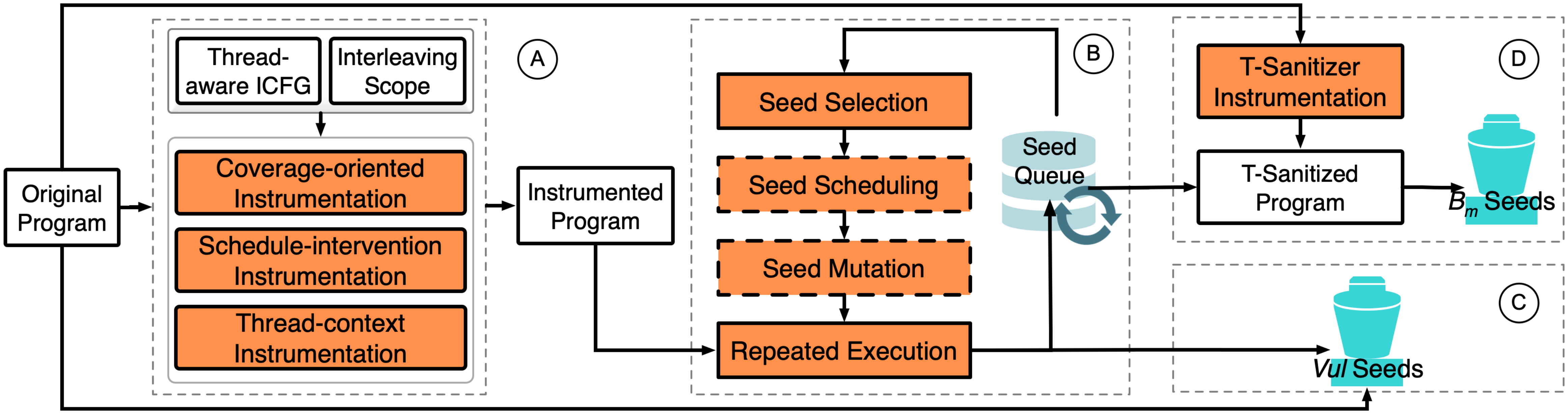


- Lacking Feedback to Track Thread-interleavings and Thread-context

  e.g., ① → ①

- Lacking Schedule-intervention Across Executions
  - e.g., SAME interleaving during fuzzing?

MUZZ Overview



Ⓐ: Static Analysis Guided Instrumentation

Ⓑ: Adaptive Dynamic Fuzzing

Ⓒ: Vulnerability Detection Analysis

Ⓓ: ThreadSanitizer Aided Concurrency-bug Revealing

Approach (1) – Static Analysis

```
int g_var = -1;
void modify(int *pv) { *pv -= 2;}                 // ⑨

void check(char * buf) {
  if (is_invalid(buf)) { exit(1); }
  else { modify((int*)buf); }
}

char* compute(void *s_var) {
  g_var += 1;                                     // ①
  g_var *= 2;                                     // ②
  if ((int*)s_var[0]<0)                           // ③
     modify((int*)s_var);                         // ④
  pthread_mutex_lock(&m);                         // ⑤
  modify(&g_var);                                 // ⑥
  pthread_mutex_unlock(&m);                       // ⑦
  return (char*)s_var;                            // ⑧
}

int main(int argc, char **argv) {
  char * buf = read_file_content(argv[1]);
  check(buf);
  pthread_t T1, T2;
  pthread_create(T1,NULL,compute,buf);
  pthread_create(T2,NULL,compute,buf+128);
  ......
}
```

Identify Suspicious Interleaving Scope (L_m)

- The statements should be executed after one of *TFork*, while *TJoin* is not encountered yet
- The statements can only be executed before the invocation of *TLock* and after the invocation of *TUnLock*
- The statements should read or write at least one of the *shared variables* by different threads

Approach (2) – Coverage-oriented Instrumentation

- Instrument more in L_m, but with certain probabilities

$$\mathrm{P_e}(f) = \min\left\{\frac{E(f) - N(f) + 2}{10}, 1.0\right\}$$

$$\mathrm{P}_s(f) = \min\{\mathrm{P_e}(f), \mathrm{Ps}_0\}$$

$$\mathrm{P}_m(f, b) = \min\left\{\mathrm{P_e}(f) \cdot \frac{Nm(b)}{N(b)}, \mathrm{P}_{m0}\right\}$$

input : target program P, and suspicious interleaving scope L_m

output : program P instrumented with M-Ins deputies

for $f \in P$ **do**
    for $b \in f$ **do**
        $L_m(b) = L_m \cap b$;
        if $L_m(b) \mathrel{!}= \emptyset$ **then**
            for $i \in b$ **do**
                if *is_entry_instr(i, b)* **then**
                    P $\leftarrow$ instrument_cov(P, i, 1.0);
                else if $i \in L_m$ **then**
                    P $\leftarrow$ instrument_cov(P, i, $\boxed{P_m(f, b)}$);
        else
            for $b \in f$ **do**
                i = get_entry_instr(b);
                P $\leftarrow$ instrument_cov(P, i, $\boxed{P_s(f)}$);

Approach (3) – Two Other Instrumentations

- Threading-context Instrumentation
  - Track *thread IDs* and *TLock*, *TUnLock*, *TJoin*
  - *Distinguish different transitions between threads*
- Schedule-intervention Instrumentation
  - Using *pthread_setschedparam* to adjust thread priority and apply *uniformly distributed random*
  - Increase thread scheduling diversities

Approach (4) – Seed Selection

input : seed queue Q_S, seed t at queue front
output : whether t will be selected in this round

```
if has_new_mt_ctx(Q_S) or has_new_trace(Q_S) then
    if cov_new_mt_ctx(t) then
        return true;
    else if cov_new_trace(t) then
        return select_with_prob(P_ynt);
    else
        return select_with_prob(P_ynn);
else
    return select_with_prob(P_nnn);
```

Prioritize to select those seeds that:

- Cover new regular traces
- Cover new thread-interleavings

Approach (5) – Repeated Execution

input : program $\mathbb{P}_o$, initial seed queue Q_S
output : final seed queue Q_S, vulnerable seed files T_C

$\mathbb{P}_f \leftarrow$ instrument($\mathbb{P}_o$) ; `// instrumentation`
$T_C \leftarrow \emptyset$;
while *True* **do**
    t $\leftarrow$ select_next_seed(Q_S) ; `// seed selection`
    $\mathbb{M} \leftarrow$ get_mutation_chance($\mathbb{P}_f$, t) ; `// seed scheduling`
    for $i \in 1 \ldots \mathbb{M}$ **do**
        $t' \leftarrow$ mutated_input(t) ; `// seed mutation`
        res $\leftarrow$ run($\mathbb{P}_f$, t', $\boxed{\mathbb{N}_c}$); `// repeated execution`
        if *is_crash(res)* **then** `// seed triaging`
            $T_C \leftarrow T_C \cup \{t'\}$; `// report vulnerable seeds`
        else if *cov_new_trace(t', res)* **then**
            $Q_S \leftarrow Q_S \oplus t'$; `// preserve "effective" seeds`

$$\mathbb{N}_c(t) = N_0 + N_v \cdot B_v, \quad B_v \in \{0,1\} \quad \Rightarrow \quad \mathbb{N}_c(t) = N_0 + \mathtt{min}\{N_v, N_0 \cdot C_m(t)\}$$

Statistics of Target Programs

ID	Project	Command Line Options	Binary Size	T_{pp}	N_b	N_i	N_{ii}	$\frac{N_{ii}-N_b}{N_b}$
lbzip2-c	lbzip2-2.5	lbzip2 -k -t -9 -z -f -n4 FILE	377K	7.1s	4010	24085	6208	54.8%
pbzip2-c	pbzip2-v1.1.13	pbzip2 -f -k -p4 -S16 -z FILE	312K	0.9s	2030	8345	2151	6.0%
pbzip2-d	pbzip2-v1.1.13	pbzip2 -f -k -p4 -S16 -d FILE	312K	0.9s	2030	8345	2151	6.0%
pigz-c	pigz-2.4	pigz -p 4 -c -b 32 FILE	117K	5.0s	3614	21022	5418	49.9%
pxz-c	pxz-4.999.9beta	pxz -c -k -T 4 -q -f -9 FILE	42K	1.2s	3907	30205	7877	101.6%
xz-c	XZ-5.3.1alpha	xz -9 -k -T 4 -f FILE	182K	8.4s	4892	34716	8948	82.9%
gm-cnvt	GraphicsMagick-1.4	gm convert -limit threads 4 FILE out.bmp	7.6M	224.4s	63539	383582	98580	55.1%
im-cnvt	ImageMagick-7.0.8-7	convert -limit thread 4 FILE out.bmp	19.4M	434.2s	128359	778631	200108	55.9%
cwebp	libwebp-1.0.2	cwebp -mt FILE -o out.webp	1.8M	56.3s	12117	134824	33112	173.3%
vpxdec	libvpx-v1.3.0-5589	vpxdec -t 4 -o out.y4m FILE	3.8M	431.6s	31638	368879	93400	195.2%
x264	x264-0.157.2966	x264 –threads=4 -o out.264 FILE	6.4M	1701.0s	38912	410453	103926	167.1%
x265	x265-3.0_Au+3	x265 –input FILE –pools 4 -F 2 -o	9.7M	78.3s	22992	412555	89408	288.9%

T_{pp}: Preprocessing time
N_b: Number of basicblocks
N_i: Number of instructions
N_{ii}: Number of MUZZ-instrumented instructions

Evaluation (1) – Seed Generation

ID	MUZZ			MAFL			AFL			MOPT		
	N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$	N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$	N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$	N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$
lbzip2-c	8056	**5127**	**63.6%**	6307	3277(+1850)	52.0%(+11.7%)	5743	2464(+2663)	42.9%(+20.7%)	6033	2524(+2603)	41.8%(+21.8%)
pbzip2-c	381	**126**	**33.1%**	340	91(+35)	26.8%(+6.3%)	272	69(+57)	25.4%(+7.7%)	279	71(+55)	25.4%(+7.6%)
pbzip2-d	1997	**297**	**14.9%**	1706	119(+178)	7.0%(+7.9%)	1650	68(+229)	4.1%(+10.8%)	1623	62(+235)	3.8%(+11.1%)
pigz-c	1406	**1295**	**92.1%**	1355	1189(+106)	87.7%(+4.4%)	1298	1098(+197)	84.6%(+7.5%)	1176	982(+313)	83.5%(+8.6%)
pxz-c	7590	**5249**	**69.2%**	5637	3401(+1848)	60.3% (+8.8%)	5357	2470(+2779)	46.1% (+23.0%)	5576	2634(+2615)	47.2% (+21.9%)
xz-c	2580	**1098**	**42.6%**	2234	767(+331)	34.3%(+8.2%)	1953	581(+517)	29.7%(+12.8%)	1845	566(+532)	30.7%(+11.9%)
gm-cnvt	15333	**13774**	**89.8%**	14031	10784(+2990)	76.9%(+13.0%)	12453	8290(+5484)	66.6%(+23.3%)	12873	8956(+4818)	69.6%(20.3%)
im-cnvt	14377	**12987**	**90.3%**	12904	10610(+2377)	82.2%(+8.1%)	9935	7634(+5353)	76.8%(+76.8%)	10203	8012(+4975)	78.5%(+11.8%)
cwebp	11383	**7554**	**66.4%**	10389	6868(+686)	66.1% (+0.3%)	9754	5874(+1680)	60.2% (+6.1%)	9803	5869(+1685)	59.9%(+6.5%)
vpxdec	28892	**25593**	**88.6%**	27735	22507(+3086)	81.2%(+7.4%)	24397	18936(+6657)	77.6%(+11.0%)	27119	20896(+4697)	77.1%(11.5%)
x264	15138	**14611**	**96.5%**	14672	13413(+1198)	91.4% (+5.1%)	13211	11801(+2810)	89.3% (+7.2%)	12427	11202(+3409)	90.1%(+6.4%)
x265	12965	10704	**82.6%**	13858	**10890**(-186)	78.6% (+4.0%)	12980	9957(+747)	76.7% (+5.9%)	13142	10154 (+550)	77.3%(+5.3%)

MUZZ has advantages in increasing the number and percentages of multithreading-relevant seeds for multithreaded programs

Evaluation (2) – Vulnerability Detection

ID	MUZZ					MAFL					AFL					MOPT				
	N_c	N_c^m	N_v^m	N_c^s	N_v^s	N_c	N_c^m	N_v^m	N_c^s	N_v^s	N_c	N_c^m	N_v^m	N_c^s	N_v^s	N_c	N_c^m	N_v^m	N_c^s	N_v^s
pbzip2-c	6	**6**	**1**	0	0	6	0(+6)	**1**(0)	0	0	0	0(+6)	0(+1)	0	0	0	0(+6)	0(+1)	0	0
pbzip2-d	15	**15**	**1**	0	0	0	0(+15)	0(+1)	0	0	0	0(+15)	0(+1)	0	0	0	0(+15)	0(+1)	0	0
im-cnvt	87	**63**	**4**	24	1	49	23(+40)	2(+2)	26	1	29	6(+57)	2(+2)	23	1	32	6(+57)	2(+2)	26	1
cwebp	19	0	0	19	1	27	0(0)	0(0)	27	1	14	0(0)	0(0)	14	1	15	0(0)	0(0)	15	1
vpxdec	523	**347**	**2**	176	2	495	279(+68)	1(+1)	216	2	393	205(+142)	1(+1)	188	2	501	301(+46)	1(+1)	200	2
x264	103	**103**	**1**	0	0	88	88(+15)	**1**(0)	0	0	78	78(+25)	**1**(0)	0	0	66	66(+37)	**1**(0)	0	0
x265	43	0	0	43	1	52	0(0)	0(0)	52	1	62	0(0)	0(0)	62	1	59	0(0)	0(0)	59	1

MUZZ demonstrates superiority in exercising more multithreading-relevant crashing states and detecting concurrency-vulnerabilities

Evaluation (3) – Concurrency-bug Revealing

ID	ℙ1								ℙ2							
	MUZZ		MAFL		AFL		MOPT		MUZZ		MAFL		AFL		MOPT	
	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m
lbzip2-c	469	1	447	1	386	1	435	1	**493**	**1**	483	**1**	421	**1**	458	**1**
pigz-c	793	1	803	1	764	1	789	1	856	**1**	**862**	**1**	727	**1**	742	**1**
gm-cnvt	93	5	79	4	45	2	55	3	**133**	**5**	83	4	54	3	57	3
im-cnvt	92	3	84	3	58	2	56	2	**118**	**3**	117	**3**	65	2	59	2
vpxdec	31	3	17	1	23	1	18	1	**42**	**3**	22	1	25	1	22	1
x264	68	8	46	6	28	4	30	5	**91**	**9**	52	6	25	4	28	4

MUZZ outperforms competitors in revealing
concurrency-bugs with fuzzer-generated seeds

Q&A

Hongxu Chen

hongxu.chen@ntu.edu.sg